OLCF-6 Benchmark DNS Code

Direct Numerical Simulation of Turbulence

Fluid turbulence is generally characterized as nonlinear, unsteady, disorderly fluctuations occurring across a vast range of scales in three-dimensions. The Pseudo-Spectral DNS (PSDNS) code is designed on top of a purpose-built 3D FFT algorithm to investigate the fundamental behavior of turbulence at very high resolution through numerical integration of the Navier-Stokes equations. As is common with 3D FFT computations, MPI communications dominate the runtime (~76%) followed by the actual FFT computations (~15%) and by the packing/unpacking operations on the MPI send/receive buffers (~5%). It is not uncommon for these three aspects of a 3D FFT algorithm to consume in excess of 96% of the overall runtime of the simulations. Consequently, the PSDNS algorithm is an excellent bellwether of system performance for all scientific fields that must address large 3D FFTs or intense, large-scale MPI communications.

As part of the scaling studies, problem sizes of 2048^3, 4096^3, 8192^3, 16384^3, and 32768^3 grid points have been evaluated for performance on Frontier with the 32768^3 case being the largest known DNS simulation to date. As we double the number of points in each coordinate dimension, we're increasing the problem size and the required computational resources by a factor of 8. Due to Frontier's unique configuration and available memory, we're able to run single precision computations for the five problem sizes on 1, 8, 64, 512, and 4096 nodes respectively; double precision computations naturally doubles the node requirements to 2, 16, 128, 1024, and 8192 nodes.

For the purpose of this OLCF-6 benchmark, a "minimalist" version of the PSDNS code was created. This benchmark PSDNS code differs from the fully-featured PSDNS code in the following ways:

1. the benchmark code is "streamlined" and does NOT include the use of passive scalar transport and Lagrangian particle transport (*these sections of code are active areas of research and have not yet been published*)
2. the benchmark code is purpose-built for GPUs, and all routines to compute the FFTs on the CPU with FFTW have been removed (*OLCF focuses on the use of GPUs*)
3. the benchmark code uses hipfft for use on both AMD and Nvidia hardware whereas the full production code uses rocfft and cufft specifically for AMD and Nvidia hardware
4. the benchmark code does not perform massive I/O

The five problems sizes were evaluated with the benchmark PSDNS code using both single and double precision. Performance data for these scaling studies are include at the end of this document.

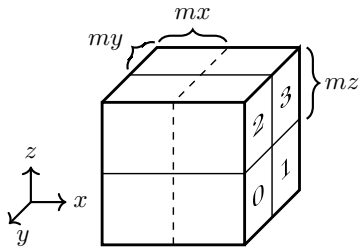
With these node-counts in mind, advancing to the next problem size (65536^3 grid points) would require a ~4x increase in overall memory for single precision computations and a ~8x increase for double precision computations and applying the full-featured PSDNS code to this problem size would require twice these amounts (e.g. ~8x and ~16x respectively).

"Minimalist" GPU-Only version of the Pseudo-Spectral DNS (PSDNS) code with 2D Domain Decomposition with the following features:

- Written in Fortran (see requirements below)
- MPI for communications from the CPU or GPU (selectable at runtime via the input file)
- OpenMP for parallelizing loops on CPUs (still a few remaining loops on the CPU)
- OpenMP Offloading to accelerate loops on the GPUs
- Extensive use of FFTs (via hipFFT) on the GPUs *and ONLY on the GPUs*
- Uses hipfort to generate the Fortran interfaces for the library calls
    - hipfort performs the switch between cuFFT (Nvidia GPUs) and rocFFT (AMD GPUs)
    - See the **Run Rules** section for guidance on modifying the code to use CUDA or OneAPI FFT instead of hipfft.

2D Domain Decomposition:

The PSDNS code uses a 2D domain decomposition strategy to divide the domain across the MPI ranks evenly.



At all times, the data is split amongst the MPI ranks in two of the three coordinate directions leaving one direction with full-length “pencils” of data for the FFT computations. Since FFTs are required in all three directions, data must be exchanged between the MPI ranks to align the data into pencils whenever FFT computations are required in a specific direction.

PSDNS solution process:

The PSDNS algorithm initializes the domain in wavenumber space and advances the solution in time with either a 2nd or 4th order Runge-Kutta (RK) algorithm. In each RK stage, we start with the solution in wavenumber space and perform the following steps:

- 3D FFT inverse transform to physical space
- form nonlinear terms in physical space
- 3D FFT forward transform to wavenumber space
- Differentiate
- advance in time ... and then repeat for each RK stage

Each 3D FFT inverse transform from wavenumber space to physical space requires:

- Perform 1D C2C FFT in the y-direction
- Pack send buffer, MPI AllToAll, unpack receive buffer
- Perform 1D C2C FFT in the z-direction
- Pack send buffer, MPIAllToAll, unpack receive buffer
- Perform 1D C2R FFT in the x-direction

Similarly, each 3D FFT forward transform from physical space to wavenumber space requires:

- Perform 1D R2C FFT in the x-direction
- Pack send buffer, MPI AllToAll, unpack receive buffer
- Perform 1D C2C FFT in the z-direction
- Pack send buffer, MPIAllToAll, unpack receive buffer
- Perform 1D C2C FFT in the y-direction

In summary, each RK stage requires 6 separate FFT computations and 4 MPI communications, and we're performing multiple RK stages every timestep of the solution process. Hence, the overall 3D FFT process consumes ~96% of the PSDNS runtime.

Structure of the repo:

1. "documentation" directory contains descriptions of the code and the benchmark cases and instructions for building and running the code
2. "scripts_frontier" and "scripts_summit" directories contains machine-specific makefiles and scripts for setting the build/run environment, building hipfort, and building the hipfft dns code
3. "makefile" is a bare-bones makefile used only for basic functions like "make clean"
4. "makefile_pieces" directory contains initialization, basic rules, and build rules used by the machine-specific files in the "scripts_frontier" and "scripts_summit" directories
5. "src" directory contains the source code
6. "extra_tools" directory contains a program that consolidates fine-grained MPI timings (see details below)
7. "benchmarks" directory contains the single and double precision results for 5 problem sizes

Requirements (this code is made specifically for GPUs):

- Fortran compiler (2003 compliant) with OpenMP Offloading
- MPI (preferably GPU-Aware)
- Hipfort from AMD ROCm github site
- either CUDA or ROCm
- See the **Run Rules** section for guidance on modifying the code to use CUDA or OneAPI FFT instead of hipfft

Steps (detailed below):

1. Build hipfort
2. Set your environment for building hipfft (same env used when running executable)
3. Set options in makefile_initialization and in the machine-specific makefile
4. Build hipfft DNS code
5. Run cases

Building hipfort:
Since the hipfft dns code is written in fortran, it needs fortran interfaces for the hip libraries. AMD provides these interfaces in hipfort along with each ROCm release on their github site. Scripts to build hipfort on Summit (with the IBM XLCUF compiler) and Frontier (with the CCE/15 FTN compiler) are included in the "scripts_frontier" and "scripts_summit" directories. These scripts are named build_hipfort_frontier.sh and build_hipfort_summit.sh respectively. In each of these scripts, the user needs to do the following:

1. Set the VERSION of hip/rocm to load and hipfort to clone, build, and install
2. Load the set of modules that will match the environment that will be used to build and run the dns code
3. Define the locations to build hipfort and then to install hipfort
4. Run the script to build hipfort (version 5.4.0 requires ~45 minutes to build on Frontier)

Setting the environment:
The environment used for building hipfort is also needed for building and for running the DNS code. At this time, setting the environment for building hipfort and the DNS code are handled separately and require the user to ensure they are the same. Future versions of this repo will unify the build scripts to use a common approach to load the necessary modules.

Scripts for Frontier (AMD EPYC CPUs and MI250x GPUs) and Summit (IBM Power9 CPUs and Nvidia V100 GPUs) are included for building and running the DNS code.:

- setUpModules_frontier.sh
- setUpModules_summit.sh

Set options in makefile_initialization and in the machine-specific makefile:
Scripts to build the code on Frontier and Summit are included (build_dns_frontier.sh and build_dns_summit.sh) in the "scripts_frontier" and "scripts_summit" directories. These scripts source the appropriate setUpModules script to load the correct modules and to set the appropriate environment variables and then use machine-specific makefiles (e.g. makefile.frontier and makefile.summit in the "scripts_frontier" and "scripts_summit" directories) to build the code.

To avoid a lengthy makefile that includes all options for multiple machines, several files are used to customize a makefile for a specific system. The following makefile "pieces" are used:

- makefile_initialization : defines macros to enable/disable sections of code
- makefile_build_rules : defines the rules for building the object files and the executable
- makefile_basic_rules : defines rules for cleaning/maintaining the directory

These three files are in the "makefile_pieces" directory. Both makefile.frontier and makefile.summit include these three files in addition to a machine-specific section. Creating a makefile.new_machine for a new system simply involves editing the machine-specific section. Alternatively, the user could create a single, large makefile with these basic files and multiple machine-specific sections.

A basic makefile that uses only makefile_basic_rules is available for quick operations such as "make clean".

Enabling or disabling features at compile time in makefile_initialization:
Several features can be easily enabled/disabled at compile time in makefile_initialization. This file is pre-set with the macros that are typically used for the PSDNS performance tests on GPUs.

Since this version of the code is purpose-made for GPUs, OpenMP **must** be enabled.

While double precision operations are predominantly used in scientific efforts, most of the PSDNS production runs use single precision operations which allows the use of twice as many grid points or half the number of nodes (depending upon your perspective) compared to using double precision operations. As such, single precision operations are the default for PSDNS, and enabling double precision operations requires the user to uncomment the appropriate line in makefile_initialization (should be obvious in the file). Test cases and timings for both single and double precision operations on Frontier are included in benchmarks/SINUSOIDAL_CASES directory.

By default, the DNS code performs MPI communications on one variable at a time for all three velocity components to stay under the 64 GB memory limit for each GCD on Frontier and the 16 GB limit on Summit.

Detailed MPI timings are available if DETAIL_MPI is enabled in makefile_initialization. This is disabled by default since it significantly increases the runtime, but it provides detailed information for all the mpi ranks. If enabled, the code writes files to the “MPI_timings” directory in your runtime directory. The “MPI_stats.F90” file in the repo’s extra_tools directory will further reduce the data MPI_timings into a more usable format.

Setting options in the machine-specific makefile:
Machine-specifc makefiles for Frontier and Summit are included in the “scripts_frontier” and “scripts_summit” directories. Compiling flags, include paths, library paths, and machine-specific options are set in the machine-specific makefiles. Users need to pay attention to the following:

1. HIP_BASE_PATH must point to your hipfort installation.
2. OTHER_INC must contain “include” paths to HIP/ROCM and/or CUDA headers
3. OTHER_LIBS must contain “lib” paths to HIP/ROCM and/or CUDA libraries
4. GPU-specific macros needed by hipfort such as __HIP_PLATFORM_NVIDIA__
5. Macros such as CRAY_CCE and USE_MAP to address non-standard compiler functions

Building the executable:
The easiest way to build the code is to use the build_dns_frontier.sh or build_dns_summit.sh script (included in the “scripts_frontier” and “scripts_summit” directories). These scripts source the appropriate setUpModules script, create the “.srcmake” directory if it doesn’t exist, performs a “make clean”, and builds the executable. Each source code file is completely preprocessed for

macros before being built. The preprocessed files are written to ".srcmake/src", and the files in ".srcmake/src" are built into *.o object files. So, if there is any confusion about whether or not a section of code is built, users can inspect the files in ".srcmake/src".

To build the code:

1. Copy/create a setUpModules.sh script in the directory immediately above "src"
2. Copy/create a makefile.machine script in the directory immediately above "src"
3. Copy/create a build_dns_machine.sh script in the directory immediately above "src"
4. Set options in makefile_pieces/makefile_initialization and makefile.machine
5. Run the build_dns_machine.sh (code usually builds in under a minute)

Running the executable:

The code requires 4 files:

1. Submission script: "**batch.sh**"
2. Parameter file: "**input**"
3. Decomposition file: "**dims**"
4. Detailed MPI timings instruction file: "**input.mpistat**"

Each of the cases in the "benchmarks/SINUSOIDAL_CASES" directory contains the files used for the benchmark runs on Frontier.

"batch.sh" file:

"batch.sh" sets the #SBATCH values, sources setUpModules.sh to set the runtime environment, sets additional env variables, creates the "MPI_timings" directory, and performs srun.

"**input" file**:

"**input"** sets the parameters for the simulation. The lines in the file alternate between text on one that describes the values on the next line. So, each "odd" line (e.g 1^{st}, 3^{rd}, 5^{th}, …) describes the contents of the "even" line (e.g. 2^{nd}, 4^{th}, 6^{th}, …) that immediately follows it. The problem size is defined by the first 4 values entered on the second line. "nc" is set to "0" for all cases. The 5^{th} entry on the second line enables GPU-Direct MPI communications and is set to "1" for all cases. "gpumpi" can be set to "0" to use CPU MPI communications.

The first entry on the fourth line of the "input" file sets the number of time steps to perform. All cases use "nsteps = 20". The second entry on the fourth line is set larger than "nsteps" for these performance tests.

The SINUSOIDAL_CASES all use "kinit = (-2, -2, -2) on the eighth line of the "input" file.

"dims" file:

The algorithm uses a 2D decomposition strategy to divide the domain amongst the MPI ranks evenly. The "dims" file contains a single line with 2 integers. The first value in the "dims" file is called "iproc" in the code, and the second value is called "jproc". For this brief discussion, let's assign N=iproc and M=jproc. If P MPI ranks are used, the code ensures P = N x M and N <= M.

The MPI AlltoAll communications are divided into “row” communications and into “column” communications. Each “row” communicator contains N MPI ranks, and each “column” communicator contains M MPI ranks. So, if N = 2, the “row” communications occur between the two GCDs on a single AMD MI250x GPU (with the highest available bandwidth), and if N <= 8, the “row” communications occur within a single Frontier node. In these cases, the “row’ communications within a node are significantly faster than the “column” communications which must exchange messages with other nodes across the broader (and slower bandwidth) network.

For the tests on Frontier, the smaller cases usually performed best with N = 2 while the larger cases performed best with N = 4 or N = 8. See the **Run Rules** section for guidance on acceptable variations.

“input.mpistat” file:
If DETAIL_MPI is enabled in makefile_initialization, then the code will be built with extra components that collect detailed timings from all MPI ranks at each timestep. Additionally, the code will read the two integers in “input.mpistat”. For simplicity, I will describe the values in reverse order. If the second number in the file is “0”, the code does **not** collect the extra information from all the MPI ranks. However, if the value is “1”, then at the end of the last timestep, the code collects and writes all the data from all the MPI ranks to files in the “MPI_timings” directory created by “batch.sh”.

This is where the first number in “input.mpistat” becomes important. For small cases, it’s usually not a problem when all MPI ranks write to file at the same, but you don’t want to do that for really large problems. This is especially true for our largest problem size that uses 32768 MPI ranks or 65536 MPI ranks for the single and double precision tests respectively; we want to avoid all the MPI ranks writing to file at the same time. Instead, we evenly divide all the MPI ranks into small groups and assign one node in each group to collect all the data from the nodes in its group and to write a single file for its group. The first number in “input.mpistat” specifies the number of nodes assigned to each small group. So, if there are 4096 MPI ranks, and the first number in “input.mpistat” is 128, then there will be 32 groups (ie. 4096/128 = 32). One rank in each group will collect the information from the 128 nodes in the group and then write all the data to file resulting in a total of 32 separate files in the “MPI_timings” directory. The “MPI_stats.F90” file in the “extra_tools” directory is used to read these 32 files and to reduce the data into a usable form.

Solutions from Frontier in benchmarks/SINUSOIDAL_CASES
A basic sinusoidal initialization is used for these cases. The cases are named for the problem size, and are computed with both single and double precision:

Name	Problem Size	Single Precision	Double Precision
2048	2048^3	1 node (N1)	2 nodes (N2)
4096	4096^3	8 nodes (N8)	16 nodes (N16)
8192	8192^3	64 nodes (N64)	128 nodes (N128)
16384	16384^3	512 nodes (N512)	1024 nodes (N1024)
32768	32768^3	4096 nodes (N4096)	8192 nodes (N8192)

For each case, 8 MPI ranks per node and 1 GCD per MPI rank are used, and approximately 42 GB of GPU memory is required for each MPI rank (with a single GCD). Since each case has 2048^3 grid points on each node, small changes in the problem size rapidly increase the memory required for the simulation. As a comparison, the full-featured version of the PSDNS code can consume nearly 60 GB of GPU memory per MPI rank (depending upon the enabled features) for similar cases and can require 8192 nodes for single precision production simulations. This "minimalist" version of the PSDNS code was chosen to minimize the system requirements for the evaluations thereby providing greater flexibility to the evaluators.

MPI message sizes and Simulation runtimes: These values are for passing one variable at a time. We're testing an approach that may allow us to pass all three variables in a single communication (with less accumulated latencies) and will add those message sizes and timings when available.

For single precision simulations while passing 1 variable at a time.

Problem Size (N^3)	**#Ranks**	**#Rows**	**#Cols**	**P2P Row Msg Size (MB)**	**P2P Col Msg Size (MB)**
2048^3	8	2	4	2048	1024
4096^3	64	2	32	2048	128
8192^3	512	2	256	2048	16
16384^3	4096	4	1024	1024	4
32768^3	32768	4	8192	1024	0.5

Runtimes for single precision simulations

Problem Size (N^3)	**#Nodes**	**FFT (sec)**	**Pack+Unpack (sec)**	**MPI (sec)**	**Other (sec)**	**Total (sec)**
2048^3	1	1.806	1.121	2.298	0.324	5.548
4096^3	8	1.912	0.800	8.155	0.416	11.284
8192^3	64	1.455	0.628	8.873	0.410	11.366
16384^3	512	2.184	0.607	10.464	0.432	13.687
32768^3	4096	3.317	0.623	11.008	0.756	15.704

Double precision simulations while passing 1 variable at a time

Problem Size (N^3)	#Ranks	#Rows	#Cols	P2P Row Msg Size (MB)	P2P Col Msg Size (MB)
2048^3	16	2	8	2048	512
4096^3	128	4	32	1024	128
8192^3	1024	2	512	2048	8
16384^3	8192	4	2048	1024	2
32768^3	65536	8	8192	512	0.5

Runtimes for double precision simulations

Problem Size (N^3)	#Nodes	FFT (sec)	Pack+Unpack (sec)	MPI (sec)	Other (sec)	Total (sec)
2048^3	2	1.405	0.842	4.842	0.385	7.473
4096^3	16	1.756	0.632	9.089	0.498	11.975
8192^3	128	1.318	0.404	9.840	0.406	11.968
16384^3	1024	1.788	0.425	10.384	0.433	13.031
32768^3	8192	2.291	0.434	15.314	0.488	18.527

Run Rules with minimal changes:
To provide a consistent comparison of performance across systems, it is best to run the cases using the input file provided with each case in the repo. However, changes should be limited to the following situations that do not alter the source code:

1. If you need a smaller problem due to memory limitations:
    a. Smaller problem sizes can be attempted by changing the values for nx, ny, and nz in the input file (*the first three values on the 2nd line of the input file*). The values for nx, ny, and nz should be identical and evenly divisible by the number of MPI ranks being used. Further, values that are pure multiples of 2 are best.
    b. Alternatively, you can spread the current cases across more MPI ranks by increasing the values in the dims file. This process was done for the double precision computations in this repo. The values for nx, ny, and nz should be evenly divisible by the number of MPI ranks being used.

2. If you can't perform MPI communications on the GPU or you simply want to investigate MPI on the CPU:
    a. Set gpumpi=0 in the input file (*last entry on the 2nd line of the input file*).

3. If you want detailed information about the MPI communications:
    a. Uncomment the line for DETAIL_MPI in the makefile_initialization file and rebuild the code. Collecting this data will increase the runtime and should not be used during formal performance testing.

Without hipfort:
If the user chooses not to use hipfort, then the user must supply their own fortran interfaces for the library functions. The user must modify the source code and makfile to remove the dependence upon hipfort and to provide instructions for the new fortran interfaces.

Without hipfort and hipFFT:
If the user chooses not to use hipFFT, then the user must modify the source code to use their desired package. Using CUDA cuFFT instead of hipFFT is a relatively straightforward change, but moving to another FFT implementation will require more work. *Regardless of the package being used, code modifications should be limited only to those changes that are absolutely required for correct function*. Results from an extensively modified code will not be considered.

Changes to the source code should be limited to the following non-critical and critical files in the src directory:

1. Non-critical to code:
    a. GPUmeminfo.F90 (this checks GPU memory usage)
    b. hipcheck.F90 (checks for hipfft success)
    c. procinfo.F90 (uses a hip function to get device info for reporting purposes)
2. Critical to code:
    a. module.F90 (where we declare variables)
    b. epfftw.F90 (fft plans and workbuffer space is set here)
    c. itransform_vel.F90 (performs hipfft Exec*** function)
    d. transform.F90 (performs hipfft Exec*** function)
    e. kxtran_gpu.F90 (performs hipfft Exec*** function)
    f. kxcomm1_gpu.F90 (performs hipfft Exec*** function)
    g. kxcomm2_gpu.F90 (performs hipfft Exec*** function)
    h. xktran_gpu.F90 (performs hipfft Exec*** function)
    i. xkcomm1_gpu.F90 (performs hipfft Exec*** function)
    j. xkcomm2_gpu.F90 (performs hipfft Exec*** function)

The files in the non-critical list contain uses of hip functionality that can be removed and/or altered without any impact on the performance of the PSDNS algorithm. However, files in the critical list must be altered with extreme care to maintain correct function of the algorithm.

For CUDA cuFFT: Since hipFFT uses interfaces similar to cuFFT's interfaces, switching to cuFFT is straightforward and requires the user to supply the fortran interfaces for the CUDA libraries.

For another FFT implementation: Moving to another FFT implementation will likely require substantial changes to the code. The majority of changes will revolve around creating the plans and setting with the work buffers in epfftw.F90. Once the FFT plans are created, modifying the rest of the critical files will involve changes required to perform a forward or inverse transform. The user will also need to provide the fortran interfaces to their FFT implementation.